Programação Orientada a Objetos

- Classes de Objetos
- Construtor e Destrutor
- Ponteiros para Objetos
- Sobrecarga de Operadores
- Membros Estáticos
- Funções e Classes Amigas
- Herança de Classes
- Métodos Virtuais e Classes Abstratas
- Templates de Classes

} 3ª Aula

Estruturas de Dados e Algoritmos
Data Structures and Algorithms
4. Programação Orientada a Objetos
1 2 9 0 FACULDADE DE CIÊNCIAS E TECNOLOGIA UNIVERSIDADE D COIMBRA
DEPARTAMENTO DE ENGENHARIA ELECTROTÉCNICA E DE COMPUTADORES

Programação Orientada a Objetos

- Classes de Objetos
- Construtor e Destrutor
- Ponteiros para Objetos
- Sobrecarga de Operadores
- Membros Estáticos
- Funções e Classes Amigas
- Herança de Classes

Herança de Classes

- Uma vantagem da programação OO é permitir criar **classes derivadas** de uma classe já existente
- A classe derivada **herda** os membros da **classe base** e permite definir **novos** membros
- Sintaxe:

```
class <classe_derivada> : <acesso1> <classe_base1>[,
    <acesso2> <classe_base2>, ...,
    <accessoN> <classe_baseN>];
```


Herança de Classes – Exemplo

```
class CPoligono {     // classe base (base/parent class)
 protected:           // visíveis em classes derivadas
   int largura, altura;
 public:              // (...)
   void inicializaValores(int, int);
};
//---------------------------------------------------
class CRetangulo : public CPoligono { // classe derivada
 public:              // (...)
   double area(void) { return ((double) largura*altura); }
};
//---------------------------------------------------
class CTriangulo : public CPoligono { // classe derivada
 public:              // (...)
   double area(void) { return (largura*altura/2.0);}
};
```

Herança de Classes – Exemplo

```
int main () {
  CRetangulo r;
  CTriangulo t;

  r.inicializaValores (3, 5);
  t.inicializaValores (3, 5);
  cout << r.area() << endl;
  cout << t.area() << endl;

  return 0;
}
```

```
15
7.5
```


Herança de Classes

- São herdados da classe base:
  - Todos os membros `public`, `protected` e `private`, exceto:
    - Construtores e Destrutor
    - Sobrecarga de operadores atribuição (=, +=, -=, *=, …)
    - Declarações de classes e funções amigas
  - Construtores e destrutor:
    - Embora não sejam herdados, a classe derivada mantém o construtor por defeito e o destrutor da classe base, que são chamados automaticamente quando um objeto da classe derivada é criado ou destruído

Herança de Classes

- Tipos de Herança:
  - A palavra reservada após os dois pontos (:) indica o nível mais acessível que os membros herdados da classe base têm na classe derivada
    - `public`: os membros da classe base mantêm a sua visibilidade na classe derivada
    - `protected`: os membros `public` passam a `protected`; os outros mantêm a sua visibilidade
    - `private`: os membros da classe base passam todos a `private` na classe derivada

Herança Múltipla

- Uma classe pode ter mais do que uma classe base:

```
class CPoligono {     // classe base 1
 protected:
   int largura, altura;
 public:  // (...)
   void inicializaValores(int larg, int alt)
   { largura = larg;   altura = alt; }
};
//-----------------------------------------
class CEscreve {      // classe base 2
 public:  // (...)
   void escreve(double x)
   { cout << fixed << setprecision(3) << x << endl; }
};
```

Herança Múltipla

```
class CRetangulo: public CPoligono, public CEscreve {
 public:
   double area(void) { return ((double) largura*altura); }
};
//--------------------------------------------------------
class CTriangulo: public CPoligono, public CEscreve {
 public:
   double area(void) { return (largura * altura / 2.0); }
};
//--------------------------------------------------------
int main() {
 CRetangulo r;
 CTriangulo t;

 r.inicializaValores(3, 5);
 t.inicializaValores(3, 5);
 r.escreve(r.area());
 t.escreve(t.area());
}
```

```
15.000
7.500
```


Programação Orientada a Objetos

- Classes de Objetos
- Construtor e Destrutor
- Ponteiros para Objetos
- Sobrecarga de Operadores
- Membros Estáticos
- Funções e Classes Amigas
- Herança de Classes
- Métodos Virtuais e Classes Abstratas

Ponteiros para a Classe Base

- Uma das vantagens de derivar classes a partir de uma classe base é que um ponteiro para uma classe derivada é compatível com o ponteiro para a classe base

- Esta característica acarreta algumas vantagens na manipulação de objetos derivados da mesma classe base, *mas*...

Ponteiros para a Classe Base

```
class CPoligono {
 protected:    //visíveis em classes...
   int largura, altura; // ... derivadas
 public:
   void inicializa(int, int);
};
//-----------------------------------
class CRetangulo : public CPoligono {
 public:
   double area(void)
  { return ((double) largura*altura); }
};
//-----------------------------------
class CTriangulo : public CPoligono {
 public:
   double area(void)
   { return (largura*altura/2.0); }
};
```

```
int main () {
 CRetangulo r;
 CTriangulo t;
 CPoligono *p1 = &r;
 CPoligono *p2 = &t;

 p1->inicializa(3, 5);
 p2->inicializa(3, 5);
 cout << r.area() << endl;
 cout << t.area() << endl;
//cout<< p1->area(); //Erro
//cout<< p2->area(); //Erro
//area() não pertence a
//CPoligono, mas… próximo slide
}
```

```
15
7.5
```

Métodos Virtuais (classes polimórficas)

- Para compatibilizar a utilização de ponteiros do tipo da classe base é necessário por vezes declarar na classe base **métodos virtuais** que podem ser redefinidos nas classes derivadas
  - São declarados usando a palavra reservada `virtual`
  - Devem ser redefinidos nas classes derivadas; se tal não for feito, é aplicado o comportamento definido na classe base
- Esta propriedade é designada por polimorfismo
  - Porque cada classe derivada pode definir «à sua maneira» o comportamento dos métodos virtuais

Métodos Virtuais

```
class CPoligono {
protected:     // visíveis em classes...
 int largura, altura;  // ... derivadas
public:
 void inicializa(int, int);
 virtual double area(void){return 0.0;}
};
//-----------------------------------
class CRetangulo : public CPoligono {
 public:
   double area(void)
   {return ((double)largura*altura);}
};
//-----------------------------------
class CTriangulo : public CPoligono {
 public:
   double area(void)
   { return (largura*altura/2.0); }
};
```

```
int main () {
 CRetangulo r;
 CTriangulo t;
 CPoligono pol;
 CPoligono *p1=&r, *p2=&t, *p3=&pol;

 p1->inicializa(3, 5);
 p2->inicializa(3, 5);
 p3->inicializa(3, 5);
 cout << p1->area() << endl;
 cout << p2->area() << endl;
 cout << p3->area() << endl;
}
```

```
15
7.5
0
```

Classes Abstratas

- Têm pelo menos um método virtual que não é definido
  - Este tipo de métodos é designado virtual puro
  - Em termos de sintaxe, adiciona-se `=0` na declaração dos métodos virtuais
- Definem conceitos gerais dos quais podem ser derivadas classes mais específicas
- Cada classe derivada define os métodos virtuais puros
- Não se pode instanciar objetos de uma classe abstrata, apenas ponteiros

Estruturas de Dados e Algoritmos
Data Structures and Algorithms
4. Programação Orientada a Objetos
1 2 9 0 FACULDADE DE CIÊNCIAS E TECNOLOGIA UNIVERSIDADE D COIMBRA
DEPARTAMENTO DE ENGENHARIA ELECTROTÉCNICA E DE COMPUTADORES

Classes Abstratas

```
class CPoligono {                    // classe abstrata
 protected:
   int largura, altura;
 public:
   void inicializa(int a, int b) {
       largura=a; altura=b;
   }
   virtual double area(void) = 0;// método virtual puro
};

CPoligono poligono;                  // declaração ilegal!

CPoligono *pontPoligono;             // permitido
```

Classes Abstratas

```
class CPoligono {  // classe abstrata
 protected:
   int largura, altura;
 public:
   void inicializa(int, int);
   virtual double area(void) = 0;
   void escreveArea(void) {
        cout << this->area() << endl;
    }   // acesso a método virtual puro
};
//-----------------------------------
class CRetangulo : public CPoligono {
 public:
   double area(void) {
     return ((double) largura*altura);
   }
};
```

```
class CTriangulo : public CPoligono {
 public:
   double area(void) {
        return (largura*altura/2.0);
   }
};
//-----------------------------------
int main () {
 CRetangulo r;
 CTriangulo t;
 CPoligono *p1=&r, *p2=&t;

 p1->inicializaValores(3, 5);
 p2->inicializaValores(3, 5);
 p1->escreveArea();
 p2->escreveArea();
}
```

```
15
7.5
```


Membros Constantes

- Atributos Constantes
  - Declarados como constantes, através da palavra `const` indicada antes do tipo do atributo
  - O seu valor não pode ser modificado após a criação do objeto
  - Inicializados após a lista de parâmetros, antes do corpo do construtor
- Métodos Constantes
  - Declarados como constantes, através da palavra `const` indicada após os parênteses da lista de parâmetros
  - Não podem modificar o valor dos atributos do objeto
  - Só podem chamar métodos constantes
- Objetos Constantes só podem chamar métodos constantes

Membros Constantes

```
class CPoligono {
 protected:
   int largura, altura;
   const int numLados;                  // atributo constante
 public:
   CPoligono(int l, int a, int nLad) : numLados(nLad)
       { largura = l;  altura = a; }
   int getLargura(void) const;          // método constante
   void setLargura(int l) const { largura = l; }
};
//-------------------------------------------------
int CPoligono :: getLargura(void) const { // mét. constante
  return largura;
}
```


Programação Orientada a Objetos

- Classes de Objetos
- Construtor e Destrutor
- Ponteiros para Objetos
- Sobrecarga de Operadores
- Membros Estáticos
- Funções e Classes Amigas
- Herança de Classes
- Membros Virtuais e Classes Abstratas
- Templates de Classes

Templates de Classes

- Um «template» (modelo) de uma classe define uma classe genérica de objetos definida em função de parâmetros que determinam o tipo de alguns dados da classe
- Definição de um «template»:

```
template <tipo_parâmetro  nome_parâmetro, ... >
```

  - `tipo_parâmetro` pode ser o nome de um tipo de dados básicos, ou a palavra-chave `class` ou `typename` para significar que `nome_parâmetro` é o nome de um tipo de dados!
- Para se obter uma classe "normal" temos que definir os vários parâmetros do «template» numa lista entre < > a seguir ao nome da classe genérica

Estruturas de Dados e Algoritmos
Data Structures and Algorithms
4. Programação Orientada a Objetos
1 2 9 0 FACULDADE DE CIÊNCIAS E TECNOLOGIA UNIVERSIDADE D COIMBRA
DEPARTAMENTO DE ENGENHARIA ELECTROTÉCNICA E DE COMPUTADORES

Templates de Classes

Exemplo simples:

```
template <class T> // template com 1 parâmetro
class ParValores { // estrutura com 2 valores do mesmo tipo
    T values[2];  // usa uma tabela
  public:
    ParValores (T first, T second) {  //construtor
       values[0] = first;
       values[1] = second;
    }
};
//-----------------------------------------------------------
// declara (e inicializa) 2 objetos de tipos diferentes
ParValores <int> obj2ints (15, 18);
ParValores <double> obj2reais (15.1, 18.2);
//-----------------------------------------------------------
```

Templates de Classes

Exemplo prático:

```
template <class T> class GereLista { // usa uma tabela
  int tamanho;         // dimensão da lista
  int indiceAtual;     // índice do próximo elem. a inserir
  T *tabela;           // ponteiro para 1º elem da lista
 public:
  GereLista(int = 100);  //construtor c/ valor por defeito
  ~GereLista();         // destrutor
  bool adicionaItem(const T &item);               // insere item
  bool itemPorIndice(T &item, int indice) const;  // le item
  bool listaVazia() const;
  bool listaCheia() const;
};
```


Templates de Classes

```
template <class T> GereLista<T>::GereLista(int max) {
  tamanho = max;                                    // construtor
  indiceAtual = 0;
  tabela = new T[tamanho];
}
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
template <class T> GereLista<T>::~GereLista() {          // destrutor
  delete [ ] tabela;
}
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
template <class T> bool GereLista<T>::adicionaItem(const T &item) {
  if (indiceAtual >= tamanho) return false;  // NOK, lista cheia
  tabela[indiceAtual] = item;                // insere item
  indiceAtual++;                             // atualiza indiceAtual
  return true;                               // OK
}
```

Templates de Classes

```
template <class T>
bool GereLista<T>::itemPorIndice(T &item, int indice) const {
   if (indice >= indiceAtual ||
       indice < 0) return false;// indice tem valor inválido
   item = tabela[indice];
   return true;
}
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
template <class T>
bool GereLista<T>::listaVazia() const {
   return (indiceAtual == 0);
}
//- - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
template <class T>
bool GereLista<T>::listaCheia() const {
   return (indiceAtual >= tamanho);
}
```


Templates de Classes

```
int main() {
 GereLista<int> listaInteiros; // com 100 inteiros por omissão
 GereLista<double> listaDoubles(10);  // com 10 elems. tipo double

 int i = 0;   double d = 0;

 // Insere números do tipo int
 while ( listaInteiros.adicionaItem(i++) );

 // Insere números do tipo double
 while ( listaDoubles.adicionaItem(d)) d+=1.5;

 if (listaInteiros.itemPorIndice(i, 3)) cout << i << endl;
 if (listaDoubles.itemPorIndice(d, 15)) cout << d << endl;

 return 0;
}
```

```
3
```

1 2 9 0 FACULDADE DE CIÊNCIAS E TECNOLOGIA UNIVERSIDADE D COIMBRA
DEPARTAMENTO DE ENGENHARIA ELECTROTÉCNICA E DE COMPUTADORES

5. Estruturas de Dados Ligadas

A partir da próxima aula…